# *Part TWO*
# Generics and Collections

# *Chapter SIX*
# Generics

### *Exam Objectives*

*Create and use a generic class.*

## Generics

Without generics, you can declare a `List` like this:

```
List list = new ArrayList();
```

Because a `List` , by default, accepts objects of any type, you can add elements of different types to it:

```
list.add("a");
list.add(new Integer(1));
list.add(Boolean.TRUE);
```

And get values like this:

```
String s = (String) list.get(0);
```

This can lead to ugly runtime errors and more complexity. Because of that, generics were added in Java 5 as a mechanism for type checking.

A generic is a type declared inside angle brackets, following the class name. For example:

```
List<String> list = new ArrayList<String>();
```

By adding the generic type to `List` , we are telling the **COMPILER** to check that only `String` values can be added to the list:

```
list.add("a"); // OK
list.add(new Integer(1)); // Compile-time error
list.add(Boolean.TRUE); // Compile-time error
```

Since now we only have values of one type, we can safely get elements without a cast:

```
String s = list.get(0);
```

It's important to emphasize that generics are a thing of the compiler. At runtime, Java doesn't know about generics.

Under the hood, the compiler inserts all the checks and casts for you, but at runtime, a generic type is seen by Java as a `java.lang.Object` type.

In other words, the compiler verifies that you're working with the right type and then, generates code with the `java.lang.Object` type.

The process of replacing all references to generic types with `Object` is called *type erasure*.

Because of this, at runtime, `List<String>` and `List<Integer>` are the same, because the type information has been erased by the compiler (they are just seen as `List` ).

Generics only work with objects. Something like the following won't compile:

```
List<int> list = new ArrayList<int>();
```

Finally, a class that accepts generics but is declared without one is said to be using a *raw type*:

```
// Raw type
List raw = new ArrayList();
// Generic type
List<String> generic = new ArrayList<String>();
```

## The Diamond Operator

Since Java 7, instead of specifying the generic type on both sides of the assignment:

```
List<List<String>> generic = new ArrayList<List<String>>();
```

We can simplify the creation of the object by just writing:

```
List<List<String>> generic = new ArrayList<>();
```

The short form on the right side is called the *diamond operator* (because it looks like a diamond).

But be careful. The above example is different than:

```
// Without the diamond operator, the raw type is used
List<List<String>> generic = new ArrayList();
```

You can only use the diamond operator if the compiler can infer the parameter type(s) from the context. The good news is that in Java 8, type inference was improved:

```
void testGenericParam(List<String> list) { }

void test() {
    // In Java 7, this line generates a compile error
    // In Java 8, this line compiles fine
    testGenericParam(new ArrayList<>());
}
```

## Generic Classes

Looking at the definition of `List` and a couple of its methods, we can see how this class is designed to work with generics:

```
public interface List<E> extends Collection<E> {
    boolean add(E e);
    Iterator<E> iterator();
}
```

We can see how a generic type is defined for classes and interfaces. It's just a matter of declaring a *type parameter* next to the class (or interface) name.

By the way, `E` is just an identifier, like a named variable. It can be anything you want. However, the convention is to use single uppercase letters. Some common letters are:

- `E` for element
- `K` for a map key
- `V` for a map value
- `T`, `U` for data types

This way, when a `List` is declared like this:

```
List<String> list = null;
```

`E` is given the value of `String`, and wherever the type `E` is defined, `String` will be used.

So generic classes give us a lot of flexibility.

For example, consider this class:

```
class Holder {
    private String s;
    public Holder(String s) {
        this.s = s;
    }
    public String getObject() {
        return s;
    }
    public void printObject() {
        System.out.println(s);
    }
}
```

There's nothing wrong with it, but it only accept objects of type `String`. What if later we need a class just like that, but that works with `Integer` types? Do we create an `Integer` version?

```
class IntegerHolder {
    private Integer s;
    public Holder(Integer s) {
        this.s = s;
    }
    public Integer getObject() {
        return s;
    }
    public void printObject() {
        System.out.println(s);
    }
}
```

Duplicate code feels and looks wrong. An `Object` version? No, thank you, we will need to add casts everywhere.

Generics help us in cases like this. Just declare a type parameter:

```
class Holder<T> {
    // ...
}
```

And the generic type `T` will be available anywhere within the class:

```
class Holder<T> {
    private T t;
    public Holder(T t) {
        this.t = t;
    }
    public T getObject() {
        return t;
    }
    public void printObject() {
        System.out.println(t);
    }
}
```

Now, when an instance is created, we just specify the type of `T` for that instance:

```
Holder<String> h1 = new Holder<>("Hi");
Holder<Integer> h2 = new Holder<>(1);
String s = h1.getObject();
```

If we don't specify a type parameter, we will be using the raw type (that uses the `Object` type):

```
Holder h3 = new Holder("Hi again");
Object o = h3.getObject();
```

If we need it, we can have more than one type parameter:

```
class Holder<T, U> {
    // ...
}
```

## Generic Methods

We can also declare type parameters in any method (not for the whole class). But the syntax is a little different, for example:

```
class Utils {
    public static <T> void print(T t) {
        System.out.println(t);
    }
}
```

This defines a method that takes an argument of type `T`. Here are two more examples of generic methods:

```
<T> void genericMethod1(List<T> list) { }
<T, U> T genericMethod2(U u) {
    T t = null;
    return t;
}
```

When a method declares its own generic type, it has to be specified before the return type (in contrast to classes, which declare it after the class name).

To call the method of the first example, you can do it normally:

```
Utils().print(10);
```

Or by explicitly specifying the type between the dot and the name of the method:

```
Utils().<Integer>print(10);
```

## Wildcards

Generics are useful in many cases, but not all. We have two main problems.

You could think that since `ArrayList` implements `List`, and because `String` is a subclass of `Object`, that something like this is fine:

```
List<Object> list = new ArrayList<String>();
```

But it doesn't compile. An `ArrayList<String>` cannot be cast to `List<Object>` because when working with generics, you cannot assign a derived type to a base type; both types should be the same (either explicitly or by using the diamond operator).

Think about it this way: a list of type `List<Object>` can hold instances of `Object` and its subclasses. In other words, the list could hold any object type, not only strings. So you could have a list of strings and integers for example, which clearly violates type safety.

But if you change the declaration to use a wildcard parameter:

```
List<?> list = new ArrayList<String>();
```

It will compile.

The unbounded wildcard type ( `<?>` ) means that the type of the list is unknown so that it can match **ANY** type.

In fact you can consider in a way `List<?>` as the superclass of all `List`s, since you can assign any type of `List`:

```
List<String> stringList = new ArrayList<>();
List<Integer> intList = new ArrayList<>();
// No problem
List<?> unknownTypeList = stringList;
// No problem either
List<?> unknownTypeList = intList;
for(Object o : unknownTypeList) { // Object?
   System.out.println(o);
}
```

Since the compiler doesn't know the type of the elements of `List<?>` , we have to use `Object` to assure there won't be any problem at runtime.

But don't think that `List<Object>` is the same as `List<?>` . It's not. With `List<Object>` the previous examples won't compile.

There's another difference. The following code won't compile:

```
List<?> list = new ArrayList<String>();
list.add("Hi"); // Compile-time error
```

Since the compiler cannot infer the type of the elements, it can't assure type safety (you can only insert null because it doesn't have a type).

To avoid this problem, the compiler generates an error when you try to modify the list. This way, when using an unbounded wildcard the list becomes **IMMUTABLE**.

This can be a problem or a benefit, depending on how you use it. This wildcard is used in arguments of methods where the code just uses methods of the generic class or from `Object` , not of a particular type, for example:

```
// You can pass any type of List here
int getSize(List<?> list) {
    return list.size();
}
```

That was the first problem. The second problem is that when working with a type parameter, we can only use methods from `Object` since we don't know the exact type of the type parameter, for example:

```
class Printer<T> {
   public void print(T t) {
      System.out.println(t.toUpperCase());// Error
      // What if T doesn't represent a String?
   }
}
```

The solution is to use the so-called bounded wildcards:

- `? extends T` (Upper-bounded wildcard)
- `? super T` (Lower-bounded wildcard)

By using these wildcards, you can relax a little the restrictions imposed by generics. This will also allow you to use some sort of polymorphism or subtyping with generics, and for that same reason, this is the trickiest part of the exam.

Let's start with the upper-bounded wildcard.

The error in the example above can be solved using the upper-bounded generic (not exactly a wildcard) this way:

```
class Printer<T extends String> {
   public void print(T t) {
      System.out.println(t.toUpperCase());//OK!
   }
}
```

`<T extends String>` means that any class that extends (or implements when working with an interface) `String` (or `String` itself) can be used as the type parameter. As `T` is replaced by `String`, it's safe to its methods:

```
Printer<String> p1 = new Printer<>(); // OK
// Error, Byte is not a String
Printer<Byte> p2 = new Printer<>();
```

The upper-bounded wildcard can also solve this problem:

```
List<Object> list =
        new ArrayList<String>(); // Error
List<? extends Object> list2 =
        new ArrayList<String>(); // OK!
```

Still, we can't modify the list:

```
list2.add("Hi"); // Compile-time error
```

The reason is the same. The compiler still can't know for sure what type will the list hold (we could add any type).

Notice then, that `List<Number>` is more restrictive than `List<? extends Number>`, in the sense that the former only accepts direct assignments of type `List<Number>`, but the latter, accepts direct assignments of `List<Integer>`, `List<Float>`, etc. For example:

```
List<Integer> listInteger = new ArrayList<>();
List<Float> listFloat = new ArrayList<>();
List<Number> listNumber = new ArrayList<>();
listNumber.add(new Integer(1)); // OK
listNumber.add(new Float(1.0F)); // OK
listNumber = listInteger; // Error
listNumber = listFloat; // Error

List<? extends Number> listExtendsNum = new ArrayList<>();
// This would cause an error
// listExtendsNum.add(new Integer(1));
listExtendsNum = listInteger; // OK
listExtendsNum = listFloat; // OK
```

Finally, we have the lower-bounded wildcard. If we have a list like this:

```
List<? super Integer> list = new ArrayList<>();
```

It means that list can be assigned to an `Integer` list (`List<Integer>`) or some supertype of `Integer` (like `List<Number>` or `List<Object>`).

This time, since you know that the list would be typed to at least an `Integer`, it's safe for the compiler to allow modifications to the list:

```
List<? super Integer> list = new ArrayList<>();
list.add(1); // OK!
list.add(2); // OK!
```

Think about it, even if the list's type is `List<Object>` , an `Integer` can be assigned to an `Object` or a `Number` (or another superclass if there were another one) for that matter.

And what types can we add to the list?

We can add instances of `T` or one of its subclasses because they are `T` also (in the example, `Integer` doesn't have subclasses, so we can only insert `Integer` instances).

So don't get confused, one thing is what can you assign and another thing is what you can add, for example:

```
List<Integer> listInteger = new ArrayList<>();
List<Object> listObject = new ArrayList<>();
List<? super Number> listSuperNum = new ArrayList<>();
listSuperNum.add(new Integer(1)); // OK
listSuperNum.add(new Float(1.0F)); // OK
listSuperNum = listInteger; // Error!
listSuperNum = listObject; // OK
```

## Generic limitations

We have talked about some of the limitations of generics, and others can be inferred from what we've reviewed, but anyway, here's a summary of all of them:

Generics don't work with primitive types:

```
// Use Wrappers instead
List<int> list = new ArrayList<>();
```

You cannot create an instance of a type parameter:

```
class Test<T> {
   T var = new T();
   // You don't know the type's constructors
}
```

You cannot declare `static` fields of a type parameter:

```
class Test<T> {
   // If a static member is shared by many instances,
   // and each instance can declare a different type,
   // what is the actual type of var?
   static T var;
}
```

Due to type erasure, you cannot use `instanceof` with generic types:

```
if(obj instanceof List<Integer>) { // Error
}
if (obj instanceof List<?>) {
    // It only works with the unbounded
    // wildcard to verify that obj is a List
}
```

You cannot instantiate an array of generic types

```
class Test<T> {
    T[] array; // OK
    T[] array1 = new T[100]; // Error
    List<String>[] array2 = new List<String>[10]; // Error
}
```

You cannot create, catch, or throw generic types

```
class GenericException<T> extends Exception { } // Error

<T extends Exception> void method() {
    try {
        // ...
    } catch(T e) {
        // Error
    }
}
```

However, you can use a type parameter in a `throws` clause:

```
class Test<T extends Exception> {
    public void method() throws T { } // OK
}
```

You cannot overload a method where type erasure will leave the parameters with the same type:

```
class Test {
    // List<Integer> and List<Integer>
    // will be converted to List at runtime
    public void method(List<String> list) { }
    public void method(List<Integer> list) { }
}
```

## Key Points

- Generics are a mechanism for type checking at compile-time.
- The process of replacing all references to generic types at runtime with an `Object` type is called *type erasure*.
- A generic class used without a generic type argument (like `List list = null;`) is known as a *raw type*.
- The *diamond operator* ( `<>` ) can be used to simplify the use of generics when the type can be inferred by the compiler.
- It's possible to define a generic class or interface by declaring a *type parameter* next to the class or interface name.
- We can also declare type parameters in any method, specifying the type before the method return type (in contrast to classes, which declare it after the class name).
- The unbounded wildcard type ( `<?>` ) means that the type of the list is unknown so that it can match **ANY** type. This also means that for example, `List<?>` is a supertype of any `List` type (like `List<Integer>` or `List<Float>` ).
- The upper-bounded wildcard ( `? extends T` ) means that you can assign either `T` or a subclass of `T` .
- The lower-bounded wildcard ( `? super T` ) means that you can assign either `T` or a superclass of `T` .

## Self Test

1. Given:

```
public class Question_6_1 {
    public static void main(String[] args) {
        Question_6_1 q = new Question_6_1();
        List<Integer> l = new ArrayList<>();
        l.add(20);
        l.add(30);
        q.m1(l);
    }
    private void m1(List<?> l) {
        m2(l); // 1
    }
    private <T> void m2(List<T> l) {
        l.set(1, l.get(0)); // 2
        System.out.println(l);
    }
}
```

What is the result?
A. `[20, 20]`
B. Compilation fails on the line marked as `// 1`
C. Compilation fails on the line marked as `// 2`
D. An exception occurs at runtime

2. Given:

```
public class Question_6_2 <T extends Number> {
    T t;
    public static void main(String[] args) {
        Question_6_2 q =
           new Question_6_2<Integer>(); // 1
        q.t = new Float(1); // 2
        System.out.println(q.t);
    }
}
```

What is the result?
A. `1.0`
B. Compilation fails on the line marked as `// 1`
C. Compilation fails on the line marked as `// 2`
D. An exception occurs at runtime

3. Which of the following declarations don't compile?
A. `List<?> l1 = new ArrayList<>();`
B. `List<String> l2 = new ArrayList();`
C. `List<? super Object> l3 = new ArrayList<String>();`
D. `List<? extends Object> l4 = new ArrayList<String>();`

4. Given

```
List<? super Number> list = new ArrayList<Object>(); // 1
list.add(new Integer(2)); // 2
list.add(new Object()); // 3
```

Which line will generate a compile-time error?
A. Line marked as `// 1`
B. Line marked as `// 2`
C. Line marked as `// 3`
D. No compile-time error is generated

Open answers page

05. Enumerations 07. Collections